Part 10

—

Connected Components

CSCI 3110 Code

Summer 2015

Now that we have the two basic graph traversal procedures, DFS and BFS, at our disposal, we can solve various basic graph problems following the approaches discussed in class. We start with connected components. For most of these problems, we develop multiple functions that produce the output in different forms because we cannot be sure how the result is to be used in a particular application. All of them will be thin wrappers around a core function. In the case of connected components, it is natural to offer a labelling of the vertices with component IDs. This is most useful if there is no need to physically split the input graph into its connected components and offers the fastest way to decide whether two vertices actually belong to the same connected component. The other alternative is to physically break up the graph:

```
newtype CC = CC Int deriving (Eq, Ord, Ix)

ccLabelling :: AdjList v vl el → V → CC
ccPartition :: AdjList v vl el → [AdjList v vl el]
```

First the labelling function. Since a DFS forest contains one tree for each connected component and the nodes of each tree are just the vertices of the corresponding connected component, computing the connected components of a graph reduces to collecting the nodes of each tree:

```
ccLabelling g = componentLabelling (gNumVertices (adjGraph g)) CC (dfs g)

componentLabelling          :: Int → (Int → a) → Forest V E → V → a
componentLabelling n l f v = array (1, n) cls ! (vIx v)
  where cvs = map (map vIx) (componentVertices f)
        cls = concat $ zipWith zip cvs (map (repeat ∘ l) [1..])

componentVertices :: Forest V E → [[V]]
componentVertices = map preOrder ∘ trees
```

We use *preOrder* to collect the vertices of each tree. *componentVertices* applies this function to all trees of the given forest and returns the resulting list of lists. Now *ccLabelling g* applies the helper function

componentLabelling to a DFS forest of g. *componentLabelling* collects the component vertices of the given forest and extracts their indices because that's what we'll have to use as array indices. It then creates a list of lists of component labels of the form $[[CC\ 1, CC\ 1\,..], [CC\ 2, CC\ 2\,..]\,..]$. This is what *map* $(repeat \circ l)\ [1\,..]$ does, given that we pass *CC* as argument l to *componentVertices*. Then it annotates each vertex with its component label using *zipWith zip cvs* $(map\ (repeat \circ l)\ [1\,..])$ and flattens the list using *concat*. What we have now is a list of pairs of vertex IDs annotated with component IDs. We now build an array mapping vertex IDs to component IDs. The function *componenLabelling n l vs* is now implemented as looking up vertex indices in this array. The reason we factored *componentLabelling* into a separate function is that we will reuse this function using a different forest partition *sccForest* to obtain a labelling function *sccLabelling* labelling the strongly connected components of g. We'll apply the same factoring to *ccPartition*.

Partitioning the graph into adjacency lists of its subgraphs is impossible using the representation we've chosen here without building the new adjacency lists completely from scratch. This takes linear time but is non-trivial, one of the reasons we offer a labelling function for situations where this is enough and thus allows us to avoid the cost of building new adjacency lists. Since we build adjacency lists from scratch anyway, we can do this rather elegantly by first collecting a *Graph* representation of each connected component and then converting each graph back to an *AdjList*. One complication is that we want deconstruction/construction to take linear time no matter the type of the vertex labels. So we have to build intermediate adjacency lists first where we know the graph labels are of an *Ix* type; at the end, we convert this back to the original labels. This in itself is not sufficient; if we want the conversion of each graph to an adjacency list representation to take linear time in the size of the graph, we need to ensure the vertex IDs aren't only taken from an *Ix* type but form a contiguous range. The array we allocate to do the conversion has size linear in the range from the smallest vertex ID to the largest vertex ID. So here's the strategy we use: First we set up an array that maps every vertex index to a new vertex index such that the vertices in each connected component have contiguous labels. Then we collect the vertices and edges of each connected component, temporarily labelling every vertex with a pair consisting of its ID and its label and using the index in the connected component as vertex ID. Finally, we convert back to every vertex having its ID as ID and its label as its only label:

$$ccPartition\ g = componentPartition\ g\ (dfs\ g)$$

```
componentPartition      :: AdjList v vl el → Forest V E → [AdjList v vl el]
componentPartition g f = adjs

  where cvs = componentVertices f
        ces = map (filter sameComponent ∘ concatMap vOutEdges) cvs

        sameComponent e = cIx (eTail e) ≡ cIx (eHead e)

        n = gNumVertices (adjGraph g)

        vIxs'  = concat $ zipWith (λvs i → zipWith (λv j → (vIx v, (j, i))) vs [1..]) cvs [1..]
        vIxA'  = array (1, n) vIxs'
        vIx' v = fst   (vIxA' ! vIx v)
        cIx  v = snd (vIxA' ! vIx v)

        cvsi = map (map mkVertex) cvs
        cesi = map (map mkEdge) ces

        gs = zipWith Graph cvsi cesi

        adjsi = map makeDirAdjListIx gs
        adjs  = map dropVertexIndices adjsi

        mkVertex v = Vertex (vIx' v)                          (adjVertexId g v, adjVertexLabel g v)
        mkEdge   e = Edge   (vIx' (eTail e)) (vIx' (eHead e)) (adjEdgeLabel g e)

        dropVertexIndices g' = g' {adjVertexId      = makeVAttr ids
                                  , adjVertexLabel = makeVAttr labs
                                  }
          where ids  = map (λv → (vIx v,fst  $ adjVertexLabel g' v)) (gVertices (adjGraph g'))
                labs = map (λv → (vIx v,snd $ adjVertexLabel g' v)) (gVertices (adjGraph g'))
```